Seminar Report

Bound Analysis of Imperative Languages

Michael Schaper
michael.schaper@student.uibk.ac.at

31 July 2012

Supervisor: Assoc. Prof. Dr. Georg Moser

Abstract

We review a method for inferring computational complexity bounds of imperative programs automatically. This report is based on an article by Gulwani, et al.. By instrumenting a program with multiple counter variables, one can establish a computational complexity bound, if bounds on those variables can be inferred and composed appropriately. We discuss this approach in more detail and comment its advantages and disadvantages.

Contents

1 Introduction

In recent years, program analysis of imperative programs has received more attention [19, 9, 4]. The automatic analysis of computational complexity seems still to be a challenge. In this report, we review an article by Gulwani, et al. [11], describing an approach for inferring computational complexity bounds for `C/C++` programs. The main idea is as follows: The program is instrumented with multiple counter variables. This variables ought to count the number of iterations of a while statement. Hence, back-edges of while statements are instrumented with an increment by one. Standard techniques in static analysis can be used to generate invariants on those variables. The technique used by the authors is based on abstract interpretation. If an invariant represents a bound on a counter variable, then it represents a bound on the number of iterations of the while statement. By composing those bounds appropriately, one can infer a bound on the total number of loop iterations of a program. To analyze the iteration on data-structures, quantitative functions are introduced. A quantitative function describes a property of a data-structure, for instance, the length of a list. Furthermore, one have to describe the effect of a method call on a quantitative function. The actual method calls in a program are then replaced by the effect of the according call. For this reason, the invariant generator has to be extended to support uninterpreted functions. A symbolic complexity bound on a program is then specified in terms of its scalar input and user-defined quantitative functions.

This report is structured as follows: In Section 2 we give an overview over the theory of abstract interpretation. We recall the main approach for generating computational complexity bounds for imperative programs in Section 3. The evaluation of our experiments are described in Section 4. We conclude in Section 5.

2 Abstract Interpretation

In this Section, we review the basic concepts of the theory of *abstract interpretation* [6, 7, 16]. The framework introduced by P. Cousot and R. Cousot is heavenly used in static analysis and plays an important role in the reviewed approach. Though, this section gives just an overview over abstract interpretation and is not important to understand the upcoming sections. The reader may skip it.

Consider an arbitrary program where its state is defined by the valuation of its variables. Suppose we could associate the set of states that are defined at some program location with the program location itself. This would be sufficient to analyze several static properties for a program, such as bound-checks for arrays. Though in general this would require to inspect all possible program executions and therefore is not computable. Abstract interpretation allows to summarize the desired property in cost of accuracy. Therefore, it relies on domains which may not be so concrete as the set of states itself. Abstract interpretation is a very general concept. Similar to the set of states one may consider all computation traces as basis for the analysis.

The underlying domain for abstract interpretation are *lattices*. Therefore, we are going to present some concepts of the lattice theory [6, 16], before describing important aspects of the theory of abstract interpretation [6, 7, 16] in more detail.

2.1 Lattice Theory

Definition 2.1. A *partially ordered set* or *poset* is a set L equipped with a *partial ordering* $\sqsubseteq$. A partial ordering is a relation $\sqsubseteq$ that is reflexive, transitive and anti-symmetric. Let $l \in L$ and Y be a subset of L. Then l is an *upper bound* (*lower bound*) of Y if for all $l' \in Y$ the relation $l' \sqsubseteq l$ ($l \sqsubseteq l'$) holds. An upper bound l_0 is a *least upper bound* (*greatest lower bound*) of Y if $l_0 \sqsubseteq l$ ($l \sqsubseteq l_0$) holds for all upper bounds (lower bounds) l of Y. We use $\bigsqcup Y$ ($\bigsqcap Y$) to denote the least upper bound (lowest greater bound) of Y. A subset Y of L is a *chain* if for all $l_1, l_2 \in Y$ the relation $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$ holds. A partially ordered set satisfies the *ascending chain condition* (*descending chain condition*) iff any infinite sequence $l_0 \sqsubseteq l_1 \sqsubseteq \cdots \sqsubseteq l_n \sqsubseteq \cdots$ ($l_0 \sqsupseteq l_1 \sqsupseteq \cdots \sqsupseteq l_n \sqsupseteq \cdots$) is not strictly increasing (decreasing).

Definition 2.2. A *complete lattice* $L = (L, \sqsubseteq, \bigsqcap, \bigsqcup, \top, \bot)$ is a partially ordered set $(L, \sqsubseteq)$ such that any subsets Y have least upper bounds ($\bigsqcup Y$) and least lower bounds ($\bigsqcap Y$). Here, $\top = \bigsqcup L$ is the *greatest element* and $\bot = \bigsqcap \varnothing$ is the *least element*.

Example 2.3. Let S be an arbitrary set and $\subseteq$ denote the subset relation. Then $L = (\mathcal{P}(S), \subseteq, \bigcup, \bigcap, S, \varnothing)$ is a complete lattice.

A more complex example is the complete lattice of integer intervals. Together with abstract interpretation, it can be used for instance, for checking accessed indices of arrays.

Example 2.4. Let $\leqslant$ denote the usual ordering on $\mathbb{Z}$ extended by $-\infty \leqslant z$, $z \leqslant \infty$ and $-\infty \leqslant \infty$, for all $z \in \mathbb{Z}$. The complete lattice of integer intervals $L = (\textsc{Interval}, \sqsubseteq, \mathsf{sup}, \mathsf{inf}, \top, \bot)$ is then defined as follows:

$$\textsc{Interval} = \{\bot\} \cup \{[z_1, z_2] \mid z_1 \leqslant z_2, z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}\}$$

$$\mathsf{inf}(int) = \begin{cases} \infty & \text{if } int = \bot \\ z_1 & \text{if } int = [z_1, z_2] \end{cases} \qquad \mathsf{sup}(int) = \begin{cases} -\infty & \text{if } int = \bot \\ z_2 & \text{if } int = [z_1, z_2] \end{cases}$$

$$int_1 \sqsubseteq int_2 \text{ iff } \mathsf{inf}(int_2) \leqslant \mathsf{inf}(int_1) \wedge \mathsf{sup}(int_1) \leqslant \mathsf{sup}(int_2)$$

Here, $\bot$ denotes the empty interval and $\top$ the interval $[-\infty, \infty]$.

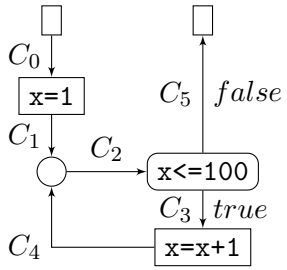
Definition 2.5. Let $f\colon L \to L$ be a monotone function on a complete Lattice $(L, \sqsubseteq, \bigsqcap, \bigsqcup, \top, \bot)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$. A fixed point l_0 is the *least fixed point* (*greatest fixed point*) if $l_0 \sqsubseteq l$ ($l \sqsubseteq l_0$) holds for all fixed points l of L.

Let L be a poset and f be a monotone function. Due to Tarski's fixed point theorem [20] we know that the set of all fixed points of f forms a complete lattice with respect to $\sqsubseteq$. Therefore, f has a least fixed point and a greatest fixed point.

2.2 Abstract Interpretation

Usually, we are interested in the least fixed point of a monotone function f. This can be obtained in an iterative manner, applying Kleene's sequence $f^n(\bot)$.

Example 2.6. This example shows the control flow graph of a single while statement. We obtain the system of equations $C_1, \ldots, C_5$. The fixed point of this system of equations represents the values x can be mapped to at C_i.



$$\begin{aligned} C_0 &= \bot & C_3 &= C_2 \cap [-\infty, 100] \\ C_1 &= [1,1] & C_4 &= C_3 + [1,1] \\ C_2 &= C_1 \cup C_4 & C_5 &= C_2 \cap [101, \infty] \end{aligned}$$

This process may not terminate if for example the sequence does not satisfy the ascending chain condition. Therefore, a *widening operator* is used, which on the one hand accelerates the computation of the fixed point and the other hand ensures convergence of the computation. Though widening may lead to inaccurate results.

Definition 2.7. An operator $\triangledown\colon L \times L \to L$ is a *widening operator* if:

(i) for all $l_1, l_2 \in L$ we have $l_1 \sqsubseteq l_1 \triangledown l_2$ and $l_2 \sqsubseteq l_1 \triangledown l_2$, and

(ii) every infinite sequence $k_0, \ldots, k_n, \ldots$ with $k_0 = l_0, \ldots, k_n = k_{n-1} \triangledown l_n$ is not strictly increasing, i.e., admits the ascending chain condition.

Example 2.8. (Continued from 2.4). We define the widening $int_1 \triangledown int_2$ for the lattice INTERVAL as follows:

(i) if $int_1 = \bot$ then int_2 ,

(ii) if $int_2 = \bot$ then int_1 ,

(iii) otherwise, if $int_1 = [i, j]$ and $int_2 = [k, l]$ then

$$[\text{if } k < i \text{ then } -\infty \text{ else } i, \text{ if } l > j \text{ then } \infty \text{ else } j \ ] \ .$$

The relation of the concrete domain to an abstract domain is often described by a *galois connection*. For example, a galois connection allows us to relate the lattice of the powerset of integers with the lattice of intervals. Likewise, we can say that we relate the concrete valuation of an integer variable to an interval over-approximating its domain.

Definition 2.9. A *galois connection* between complete lattices is a quadruple (L, α, γ, M) such that L and M are complete lattices, $\alpha \colon L \to M$ and $\gamma \colon M \to L$ are monotone functions, satisfying

$$\forall l \in L, : l \sqsubseteq \gamma(\alpha(l)) \,, \text{and } \forall m \in M \colon m \sqsupseteq \alpha(\gamma(m)) \,.$$

Functions α, γ are usually called *abstraction function* and *concretisation function*, respectively.

The first condition implies that we may lose precision if we abstract an element of the concrete domain and then concretize it again. The second condition implies that there are no new elements if we concretize an element of the abstract domain and then abstract it again.

Example 2.10. Let L be the lattice of the powerset of integers, M be the lattice of integer interval and α, γ be defined as follows:

$$\alpha(l) = [\min(l), \mathsf{max}(l)] \qquad \gamma(m) = \{x \mid x \in [z1, z2]\}$$

Then (L, α, γ, M) is a galois connection and $\gamma(\alpha(\{2, 3, 7\})) = \{2, 3, 4, 5, 6, 7\}$ and $\alpha(\gamma([2, 7])) = [2, 7]$.

For the analysis of numerical properties different abstract domains have been established. We already have described the integer interval domain. A more sophisticated domain is the abstract domain of the convex polyhedra [8]. A closed convex polyhedron is the solution of a set of linear inequalities $c_1 * x_1 + \cdots + c_n * x_n \leqslant b$. Intuitively the variables $x_1, \ldots, x_n, b$ represent variables of a program. The solution of a set of linear inequalities represents the domain of the variables. In comparison to the interval domain the polyhedra domain is more precise and allows us to relate the variables to each other. Though the analysis is not as efficient as for other domains.

The polyhedra domain is often used in static analysis and allows to generate invariants of the domains of the variables. We will see in the next chapter, how this can be exploited to infer upper bounds of the computational complexity of a program.

3 The Speed Method

In this section, we recall the main concepts and definitions of [11]. The ideas have been implemented in the SPEED tool, as a plug-in for the Microsoft Phoenix compiler infrastructure [1]. The tool computes symbolic complexity bounds for C/C++ programs. As a preprocessing step the tool extracts a slice of the program of interest with respect to statements affecting the number of loop iterations.

3.1 Introduction

Already in [7] P. Cousot and R. Cousot state that abstract interpretation can be used for analyzing the complexity of a program using an imaginary counter. The idea is very simple: We take a single counter which is initialized to zero at the beginning of a procedure and incremented by one at every program location describing a back-edge of a loop. If we can compute the final value of the counter due to abstract interpretation, we obtain a bound on the total number of loop iterations. Using the polyhedra domain, as introduced in the previous section, seems reasonable. If we can generate an invariant $d_1 * x_1 + \cdots + d_n x_n \geqslant c$, where $x_1, \ldots, x_m$ represents the input parameter of a program and c the counter variable, then we obtain an upper bound on c in terms of the program's input. The process of adding such a counter variable is called *counter instrumentation*. In general, computing non-trivial bounds for an arbitrary program is not possible. Gulwani, et al. developed some strategies to obtain a more practical analysis. They identify and address four general problems for inferring complexity bounds of a program:

(i) The presence of disjunctive and non-linear bounds,

(ii) the fact that for many problems even termination is hard to prove,

(iii) precise bounds are desired, and

(iv) the iteration over user defined data-structures.

Note that abstract interpretation used together with the polyhedra domain is a well-researched technique in static analysis. The first problem is strongly related to the use of the polyhedra domain since polyhedra are defined by conjunctions of linear inequalities. Though it seems to be a general challenge in program analysis to handle disjunctive invariants [18]. It seems quite sensible considering a simple program consisting of a while loop with an additional if-statement in its body. In static analysis it may require that both the execution of the then branch and the execution of the then branch have to be described using a single property. In case of the polyhedra domain, a conjunction of linear inequalities. The authors handle the first two problems by instrumenting the program with multiple counters. These counters can be initialized and incremented at different program locations. If the invariant generator is capable of inferring linear bounds for each counter then the total number of loop iterations can be inferred by composing the bounds appropriately.

The SPEED tool was designed to establish precise bounds. Here, precise means that both the computational complexity as well as the constant factors are of interest. The bounds can vary depending on the instrumentation of the counter variables. The authors developed and implemented an algorithm returning an "optimal" complexity bound.

The fourth problem addresses the problem of inferring bounds on iterations over data-structures, such as lists and trees. Instead of analyzing the programs directly, the tool requires the user to define *quantitative functions*, which are numeric functions describing properties of the data-structure, together with annotations, which describe the effect on a quantitative function calling a data-structure method. By extending the invariant generator to support uninterpreted functions, the SPEED tool is capable of describing linear bounds on data-structure iterations in terms of quantitative functions.

If the analysis of a program is successful, we obtain a bound on the total number of loop iterations in terms of its scalar input and quantitative functions.

3.2 Proof-Structure

In this subsection we are going to present the overall methodology of [11]. In the following let

- S denote a set of counter variables $c_1, \ldots, c_n$,
- M denote a mapping from back-edges $q_1, \ldots, q_m$ to variables in S,
- G denote a directed acyclic graph (DAG) ranging over $S \cup \{r\}$, where r denotes a dedicated root node, and
- B denote a mapping from back-edges to bounds.

Instrumentation is the process of adding the counter variables to a program P. The instrumentation of a program P is formally defined as follows:

Definition 3.1. Let P a program. Then $\texttt{Instrument}(P, (S, M, G))$ denotes the program obtained from P and instrumenting

- back-edge q of P with an increment by one of counter c, if $M(q) = c$,
- back-edge q of P with an initialization to zero of counter variable c', if $(c, c') \in G$ and $M(q) = c$,
- the procedure entry of P with an initialization to zero of counter variable c, if $(r, c) \in G$.

If $(c, c') \in G$, c' is initialized to zero when c is incremented. We say that counter c' depends on counter c. By initializing c' to zero, the problem for the invariant generator is simplified. This have to be considered when computing the bounds. Intuitively, the bound associated to c' represents a bound for a single iteration with respect to the back-edge associated with c. This can be compared with a nested for statement, where the inner loop is initialized in each iteration of the outer loop.

Example 3.2. Let P be the program depicted in Figure 1a and let S, M and G be defined as follows:

$$S = \{c, d\} \qquad M = \{q \mapsto c, r \mapsto d\} \qquad G = \{(r, c), (r, d)\}\,.$$

The program obtained by `Instrument`$(P, (S, M, G))$ is depicted in 1b.

```
disjunctive(int x0,z0,n)

  int x=x0; int z=z0;
  while(x<n)
    if(z>x)
      x++;

    else
      z++;
```

(a) `disjunctive.c`

```
cdisjunctive(int x0,z0,n)
  int c=0;  int d=0;
  int x=x0; int z=z0;
  while(x<n)
    if(z>x)
      x++;
q     c++;
    else
      z++;
r     d++;
```

(b) `cdisjunctive.c`

Figure 1: The instrumentation of `disjunctive.c`.

Definition 3.3. Let P be a program and $(P, (S, M, G))$ define its instrumented version. A *proof-structure* for P is a quadruple (S, M, G, B), such that the invariant generation tool can establish a bound $B(q)$ on counter variable $M(q)$ at back-edge q in instrumentation $(P, (S, M, G))$ for all back-edges q in P.

Given some program instrumentation, the proof-structure associates a bound on each back-edge q with respect to the counter which is incremented at q. Note that the bounds on the counter variables are linear. If a proof-structure can be established, we can compute a bound on the total number of loop iterations.

Example 3.4. (Continued from Example 3.2). The invariant generator tool can establish following invariants:

$$B = \{q \mapsto n - x0, q2 \mapsto n - y0\}\,.$$

Therefore, (S, M, G, B) is a proof-structure of P.

Definition 3.5. Let (S, M, G, B) be a proof-structure for P. The upper bound on the total number of loop iterations U is then defined as follows:

$$U := \sum_{c \in S} \mathsf{totalbound}(c) \tag{1}$$

$$\mathsf{totalbound}(r) := 0 \tag{2}$$

$$\mathsf{totalbound}(c) := \mathsf{max}\left(\{0\} \bigcup \{B(q) \mid M(q) = c\}\right) \times \left(1 + \sum_{(c', c) \in G} \mathsf{totalbound}(c')\right) \tag{3}$$

Equation (1) states that we sum up the total bounds of all counter variables. Equation (2) represents the base-case of the recursive call to totalbound. Note that a single counter variable can be incremented at multiple back-edges. For example, if we consider two sequent while statements. In this case, the invariant generator may be able to relate the bounds for those back-edges itself. Taking the maximum of those bounds is a safe approximation. This behavior is represented by the first factor of equation (3). The second factor sums up the total bounds for all counters on which the corresponding counter depends, in a recursive manner. Recall that if $(c', c) \in G$, then counter c depends on c'. Per definition c is initialized to zero on a back-edge q, whereas c' is incremented at q. Thus multiplying the bound generated in the first part with the total bound generated in the second part safely approximates the total number of loop iterations of the corresponding counter variable.

Example 3.6. (Continued from Example 3.4). Since, (S, M, G, B) is a proof-structure of P we can compute an upper bound U on the total number of loop iterations of P.

$$\begin{aligned} U =& \mathsf{totalbound}(c) + \mathsf{totalbound}(d) \\ =& \mathsf{max}(0, n - x0) \times (1 + \mathsf{totalbound}(r)) + \\ & \mathsf{max}(0, n - y0) \times (1 + \mathsf{totalbound}(r)) \\ =& \mathsf{max}(0, n - x0) + \mathsf{max}(0, n - y0) \end{aligned}$$

We refer to Section 4 to present how this construction handles disjunctive and non-linear bounds in more detail.

3.3 Counter-Optimal Proof-Structure

Given a program P. In the general case, a proof-structure and thus the upper bound U for P may not be unique. For example, given two sequent while statements we may be able to infer a bound using a single counter. If this is the case, we could also infer a bound using two counters. This raises the question, how a "optimal" proof-structure can be obtained. The authors observed that a minimal number of counters and a minimal number of dependencies leads usually to better results. The observations are not surprising when inspecting Definition 3.5 for computing upper bounds. Though, it may be the case that an invariant generation tool could generate better bounds using additional counters and dependencies for an arbitrary program. The authors developed an algorithm to construct a *counter-optimal* proof-structure, i.e., a proof-structure with a minimal number of counter variables and dependencies. Here, we restrict to the general idea of the algorithm. For more details see [11]:

Let S, M, G and B be empty. As long as a bound for some back-edge q, undefined in M, can be inferred, the following steps are performed: First, the algorithm tries to re-use an existing counter variable c in S for some back-edge q that is undefined in M. Therefore the algorithm instruments P with an increment of c at back-edge q. S, M, G and B are updated correspondingly, if the invariant generation tool can infer bounds for all defined back-edges in M

and q. Otherwise, a fresh counter variable c' with respect to S is introduced and P instrumented with an increment of c' at back-edge q. Furthermore, dependencies to all other variables in S and the root node r are added. The algorithm fails here, if the invariant tool can not infer bounds for all defined back-edges in M. If this is not the case, then all dependencies of c' are removed that preserve the generation of the already established bounds. If the algorithm does not fail, it returns a counter-optimal proof-structure $(P, (S, M, G, B))$.

3.4 Iteration over Data-Structures

We already have addressed three of the four problems introduced in Section 3.1. The last one remaining is how to handle iterations over data-structures such as lists and trees. Gulwani, et al. have chosen an indirect way to address the problem. They do not perform an analysis of the heap but require the user to define quantitative functions and the effects of method calls on it. A quantitative function represents a numerical property of the data-structure. For example the length of the list. It is also possible to define multiple quantitative functions such as the height of the tree and the number of its nodes. Additionally, the user defines the effects on the numerical properties in terms of a linear expression for each method manipulating the data. The quantitative functions are treated as uninterpreted functions. Therefore, it is necessary to extend the linear invariant generation tool with support for uninterpreted functions. In [12] Gulwani and Tiwari have described how to combine abstract interpreters automatically. The method is based on the article of Nelson and Oppen for combining decision procedures [15].

Example 3.7. This example displays how quantitative functions for a single-linked list can be defined. The example is taken from [11]. First, we define some quantitative functions on lists:

$$\begin{aligned} \texttt{Len}(L) &:= \text{length of list } L \\ \texttt{Pos}(e, L) &:= \text{position of element } e \text{ in list } L \end{aligned}$$

Next, we define effects on the quantitative functions for some list operations:

$$\begin{aligned} e = L.\texttt{Head}() \quad &:= \texttt{Assume}(e = \texttt{null} \Rightarrow \texttt{Len}(L) = 0); \\ &\quad \texttt{Assume}(e \neq \texttt{null} \Rightarrow \texttt{Len}(L) > 0);\ \texttt{Pos}(e, L) \\ b = L.\texttt{IsEmpty}() \quad &:= \texttt{Assume}(t = \texttt{true} \Rightarrow \texttt{Len}(L) = 0); \\ &\quad \texttt{Assume}(t = \texttt{false} \Rightarrow \texttt{Len}(L) > 0) \\ e = L.\texttt{GetNext}(f) \quad &:= \texttt{Pos}(e, L) = \texttt{Pos}(f, L) + 1; \\ &\quad \texttt{Assume}(0 \leqslant \texttt{Pos}(f, L) < \texttt{Len}(L)) \\ L.\texttt{RemoveHead}() \quad &:= \texttt{if}(\texttt{Len}(L) > 0)\{\texttt{Len}(L) = \texttt{Len}(L) - 1; \\ &\quad \texttt{Pos}(e', L) = \texttt{Pos}(e', L) - 1\} \end{aligned}$$

Here e' is a free variable and is used to represent the new positions of the elements of list L after calling method $L.\texttt{RemoveHead}()$.

We consider a program iterating over a list:

$$\texttt{for}(e = f; e \neq \texttt{null}; e = L.\texttt{GetNext}(e));$$

The invariant generator of the SPEED tool can establish following invariant: $c = \texttt{Pos}(e, L) - \texttt{Pos}(f, L) \wedge \ \texttt{Pos}(e, L) \leqslant \texttt{Len}(L)$. Simplifying it, returns $c \leqslant \texttt{Len}(L) - \texttt{Pos}(f, L)$.

Similarly we can describe a program deleting all entries of a list:

$$\texttt{for}(; !L.\texttt{IsEmpty}(); L.\texttt{RemoveHead}());$$

We obtain, $c = \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) \wedge \texttt{Len}(L) \geqslant 0$ and $c \leqslant \texttt{Old}(\texttt{Len}(L))$. Here, Old is used to represent the original length of the list.

In the extended version of the paper the authors provide the quantitative functions and the annotations for lists, list of lists, trees and bit-vectors [10].

An important property of this approach is that you do not have to analyze heap operations directly in terms of heap size or heap shapes. Moreover, since the result is returned in terms of quantitative functions, one circumvents to define the notion of complexity for heap operations. The approach seems to be very flexible and intuitive since multiple quantitative functions can be defined. For example, the complexity for traversing a tree can be defined with respect to the number of nodes whereas the complexity for searching an element can be defined with respect to the height of the tree. The biggest disadvantage may be that the analysis is not fully automatic anymore. One have to define its own quantitative functions for a new data-type. This may not be possible if the source code is not available. Furthermore, results are obtained in terms of the provided specification. This may be an additional source of an error.

3.5 Inter-Procedural Analysis

Until now, we have only considered the analysis of a single procedure. The method presented in [11] also allows to establish bounds when considering procedure calls. We assume that a procedure call does not have any effect on the procedure calling it.

First, we consider the non-recursive case. In the non-recursive case there are two main challenges to handle:

(i) the bound or the cost of calling a procedure have to be considered when computing the bounds of the calling procedure, and

(ii) the bound of the called procedure have to be related to the input.

Challenge (i) is managed in a very general way. The method described in Section 3.2 is used to infer the total number of loop iterations. This can be generalized for inferring the total cost of a procedure given a cost metric mapping atomic statements, such as procedure-calls, to a cost. This way, a bound for the total number of executed instructions or the total resource consumption can be inferred. Let q be a back-edge at location l and $M(q) = c$. The cost for

q, denoted $||q||$, is then defined by the maximum of the costs of any path that ends in l and starts from any counter initialization location l' of c, where c is not initialized between l' and l. The cost of a path is the sum of the costs of all statements and the cost for calling procedures. The cost of a program P can then be computed as follows:

$$||P|| := \sum_{c \in S} (1 + \mathsf{totalbound}(c)) \times \mathsf{max}\,\{||q|| \mid M(q) = c\}\ .$$

Challenge (ii) addresses the problem that the arguments of the called procedures have to be related with the input of the calling procedure. Let $Q(y_1, \ldots, y_n)$ be a program, and $||Q||$ be its cost in terms of its parameters $y_n, \ldots, y_n$. Let $P(x_1, \ldots, x_m)$ be the procedure invoking $Q(v_1, \ldots, v_n)$. The cost of calling Q, denoted $||\texttt{call}\ Q(v1, \ldots, v_n)||$, can be obtained by existential elimination of the formula

$$\exists V \colon t \leqslant e \wedge \phi\,,$$

where e is obtained by replacing parameters $y_1, \ldots, y_n$ with arguments $v_1, \ldots, v_n$ in $||Q||$, ϕ is an invariant generated relating arguments $v_1, \ldots, v_n$ with input $x_1, \ldots, x_n$, V are the variables occurring in e and ϕ but not occurring in P, and t is a fresh variable.

Note that $\mathsf{totalbound}(c)$ is computed independently from its cost for some counter variable c. This provides some modularity for the analysis of the non-recursive case. For example, a different cost may be obtained replacing procedure calls within a program. Though it may be necessary to compute a bound with respect to the programs input as addressed in challenge (ii).

Example 3.8. Suppose $||\texttt{double}(n)|| = \mathsf{max}(0, 2 * n)$, our cost function measures the total number of loop iterations and we want to compute the cost of the `loop` program:

```
loop(int n)
  int c=0;
  int x=0;
  while(x<n)
    x++;
    double(x);
    c++;
```

Then we obtain:

$$\begin{aligned} ||\texttt{call double}(x)|| &= \exists x \colon t \leqslant \mathsf{max}(0, 2 * x) \wedge x \leqslant n \\ &= t \leqslant \mathsf{max}(0, 2 * n) \\ ||\texttt{loop}|| &= (1 + \mathsf{totalbound}(c)) \times (\mathsf{max}(0, 2 * n)) \\ &= (1 + \mathsf{max}(0, n)) \times \mathsf{max}(0, 2 * n) \end{aligned}$$

Second, we consider the recursive case. The general idea is to use a global counter to count the number of recursive calls. Only a few changes have to be made to adapt the overall methodology for recursive calls. Let $P_1, \ldots, P_n$

be mutually recursive procedures. For each P_i a procedure P_i' is defined. Procedure P_i' serves as entry point and is obtained by copying the parameter of P_i $(x_1, \ldots, x_n)$ into global parameters $(x_1', \ldots, x_n')$ and then calling P_i. Now all procedures P_i and P_i' are put together in a module. The definitions for instrumentation and proof-structure as introduced in Section 3.2 have to be modified slightly. If $(r, c) \in G$, then c is initialized at the entry point of P_i'. Furthermore, the proof-structure now requires that M map back-edges in all the procedures and the locations immediately before any recursive procedure call to some global counter variable. The invariant generator is supposed to generate invariant in terms of global parameter $(x_1', \ldots, x_n')$. Using the same process as for the non-recursive calls, the costs for the recursive calls can be obtained after computing a proof-structure. Note that the invariant generator has to provide support for inter-procedural analysis. This is usually established by a two phased approach. In the first phase a summary of a procedure is generated, relating the inputs of a procedure to the outputs. In the second phase this summary can be used as transfer functions for procedure calls.

4 Experimental Evaluation

In this section, we are going to present the evaluation of our experiments. The main goal of the experiments are to get a better understanding of the invariant generation and the concepts presented in Section 3. The SPEED tool is part of the Microsoft Phoenix compiler infrastructure [1], which was nowhere to find for download. Therefore, a different invariant generator have been chosen for the experiments. Since we have not found another tool that provides support for uninterpreted functions, we restrict our experiments to the numerical examples of [11].

We decided to use the InvGen [13] tool, an automatic linear arithmetic invariant generator for imperative programs: The InvGen tool is easy to use and provides everything to analyze programs written in the C language. Similar to other tools, e.g. StInG [2], InvGen operates on an intermediate representation rather than on the source code directly. Fortunately, the tool provides a front-end that takes a procedure written in the C language and returning the transition relation of the program. In comparison to the SPEED tool InvGen uses a constraint-based approach to generate invariants [5], combining it with static and dynamic program analysis. Figure 2 depicts a C program with a single loop instruction and its instrumented version. Running the invariant generator

```
simple(int n)

  int x=0;
  while(x<n)
    x=x+2;
```

(a) simple.c

```
csimple(int n)
  int c=0;
  int x=0;
  while(x<n)
    x=x+2;
    c++;
```

(b) csimple.c

Figure 2: The simple.c program and its instrumented version csimple.c .

on its instrumented version returns the invariant $x = 2 * c \wedge c \geqslant 0$. Though, the result is obviously correct it does not return a bound on the counter variable c. In fact, we can not establish a bound on the counter variable with respect to the input. Consider input n to be a negative integer. Then n can not be a bound for the counter variable c, since c is at least zero. In this case, we have to strengthen the assumptions for the invariant generator. We know that the counter variable is only incremented if the loop condition holds. Therefore, we assume that the condition $x < n$ holds, when entering the loop. This is illustrated in Figure 3. Now, we obtain $n - 2 * c \geqslant -2 \wedge n \geqslant 1 \wedge x = 2 * c \wedge c \geqslant 0$ as invariant. Reformulating the first constraint yields the desired bound on the counter variable: $1/2 * n + 1 \geqslant c$. Recall Definition 3.5 for computing an upper bound on the total number of loop iterations. Making this assumption corresponds of using the max operator when computing the bound. We obtain $\mathsf{max}(0, 1/2 * n + 1)$.

Recall the program in Figure 1b depicting the instrumented version of a program with a disjunctive invariant. Using a single counter variable, we will

```
csimple2(int n)
  int c=0;
  int x=0;
  assume(x<n);
  while(x<n)
    x=x+2;
    c++;
```

Figure 3: The augmented version of `csimple.c`.

fail to establish a bound, since we can not establish a bound on variable z. The key observation of the program is, that the body of the then branch and the body of the else branch does not affect each other. In this case, we can analyze the bounds separately. Once assuming the condition $x < n$ does hold, and once assuming the condition does not hold during the execution of the program. Figure 4 depicts the case where the condition holds. We obtain

```
cdisjunctive(int x0,z0,n)
  int c=0;  int d=0;
  int x=x0; int z=z0;
  assume(x<n);
  while(x<n)
    assume(z>x);
    if(z>x)
      x++;
      c++;
    else
      z++;
      d++;
```

Figure 4: The analysis of the `cdisjunctive.c` program.

following invariants: $x - n \leqslant 1 \wedge x0 - n \leqslant 0 \wedge c = -(x0) + x \wedge x0 - x \leqslant 0$ and $x - n \leqslant 1 \wedge x0 - n \leqslant 0 \wedge z0 = z - d \wedge x0 - x \leqslant 0 \wedge d \geqslant 0$. Taking the assumptions into account and reformulating the constraints appropriately, we obtain $c < n - x0$ and $d < n - z0$.

Our next example shows, that we have to be careful when adding additional assumptions. Consider Figure 5 depicting a program with a non-linear bound. It looks tempting to use the same approach as for the previous example. But, here the variable y is modified in both branches and we are not allowed to consider the branches separately. Here, we can simplify the problem for the invariant generation tool by initializing counter variable c to zero in the else branch. We obtain following invariant: $n - d \geqslant -1 \wedge n \geqslant 1 \wedge x = d \wedge d \geqslant 0 \wedge y = c \wedge c \geqslant 0$. The first constraint already defines the bound for counter variable d. To obtain a bound for c we take the assumption $y < m$ into account. This is now valid since setting c to zero in the else branch allows us to analyze the then branch with respect to the analysis of counter variable c separately. We obtain $\mathsf{max}(0, n) + \mathsf{max}(0, m) \times \mathsf{max}(0, n)$ as upper bound.

```
simplemultipledep(int n,m)

  int x = 0; int y = 0;

  while(x<n)
    if(y<m)
      y++;

    else
      y=0;
      x++;
```

(a) `simplemultipledep.c`

```
csimplemultipledep(int n,m)
  int c=0; int d=0;
  int x=0; int y=0;
  assume(x<n);
  while(x<n)
    if(y<m)
      y++;
      c++;
    else
      y=0;
      x++;
      d++; c=0;
```

(b) `csimplemultipledep.c`

Figure 5: The analysis of the `simplemultipledep.c` program.

The results for all examples from [11], including only numerical expressions, could be reproduced using InvGen. Though, it was necessary to induce and apply additional assumptions. As explained above, these introduced assumptions are strongly related to the method of computing the bounds as presented in Section 3. It is not totally clear how this is actually handled by the invariant generator of the SPEED tool. But, as already the first examples in our experiments (cf. Figure 2) shows, it is necessary to introduce additional assumptions.

All the examples in [11] consider only constants or input arguments as bounds within the while condition. Therefore, the question arises how the invariant generator behaves when a condition is described by a variable which has been altered by side conditions. This quested scenario is depicted in Figure 6a. It

```
sequent(int n)
  int x=0; int y=0;
  while(x<n)
    x++;
    y+=2;
    c++;
  x=0;
  while(x<y)
    x++;
    c++;
```

(a) `sequent.c`

```
exp(int x)
  int y=0; int r=1;
  while (x>0)
    y=r;
    r=0;
    while(y>0)
      r+=2;
      y--;
    x--;
```

(b) `exp.c`

Figure 6: The `sequent.c` and `exp.c` program.

turns out that the InvGen invariant generator is able to find bounds if two counter variables are used. We obtain $\mathsf{max}(0, n) + \mathsf{max}(0, 2 * n)$.

The program depicted in Figure 6b has an exponential complexity behavior. Not surprising, we could not establish a bound using this method.

5 Conclusion and Future Work

In this report we have reviewed an approach by Gulwani, et al., for inferring computational complexity bounds for imperative programs. The experiments with the InvGen invariant generator tool have shown that additional reasoning and therefore some experience with invariant generation is necessary to use this approach to the fullest extent. In comparison to other methods, e.g. [21], the one described here is one of the few methods that addresses the analysis of user defined data-structures. Therefore, the notion of quantitative functions were introduced. Due to this additional requirement, the method is not fully automatic anymore. Furthermore, sophisticated invariant generators supporting uninterpreted functions and inter-procedural analysis are necessary.

Falke and Kapur have used a term rewriting approach for termination analysis of imperative programs [9]. Therefore, they introduced the notion of a $\mathcal{PA}$-based term rewriting system (TRS). A $\mathcal{PA}$-based TRS is a constrained TRS, where constraints are quantifier free formula from Presburger arithmetic expressing relations on program variables. This extension seems reasonable for the analysis of imperative programs, since arithmetic expressions in the program can be transformed directly using constraints. Results in complexity analysis of TRSs show that the complexity of TRSs can be related to the computational complexity of the functions computed by the corresponding TRSs [3]. At the moment it is not clear how the extension affects the results. Moreover, the analysis is restricted to programs without user defined data-structures. It is not trivial to transform heap operations in term rewriting rules. Therefore an additional abstraction layer as presented in [4, 17, 14] seems necessary.

References

[1] Microsoft Phoenix Compiler Infrastructure. `http://research.microsoft.com/phoenix`.

[2] StInG: Stanford Invariant Generator. `http://www.cs.colorado.edu/~srirams/Software/sting.html`.

[3] M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21th RTA*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[4] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *LNCS*, pages 17–37, 2010.

[5] M. A. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. of 15th CAV*, pages 420–432, 2003.

[6] P. Cousot. Abstract interpretation, Feb.–May 2005. MIT course 16.399, `http://web.mit.edu/16.399/www/`.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4th POPL*, pages 238–252, 1977.

[8] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proc. of 5th POPL*, pages 84–96, 1978.

[9] S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. of 22nd CADE*, pages 277–293. Springer-Verlag, 2009.

[10] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. Technical Report MSR-TR-2008-95, Microsoft Research, 2008.

[11] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of 36th POPL*, pages 127–139, 2009.

[12] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. of PLDI'06*, pages 376–386, 2006.

[13] A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proc. of 21st CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[14] G. Moser and M. Schaper. A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems. *CoRR*, abs/1204.1568, 2012.

[15] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[17] C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. of 21th RTA*, pages 259–276, 2010.

[18] R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Proc. of 23rd CAV*, pages 703–719, 2011.

[19] F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of 7th FTfJP*, 2005.

[20] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[21] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Proc. of 18th SAS*, pages 280–297, 2011.